

爱创课堂前端培训

JS 基础

第 5 天课堂笔记（本课程共 8 天）

班级：北京前端训练营 7 期

日期：2017 年 5 月 15 日

爱创课堂官网：www.icketang.com

目录

JS 基础.....	1
目录.....	1
复习.....	2
一、 函数.....	3
1.1 递归函数.....	3
1.2 变量的作用域.....	3
1.3 局部变量和全局变量.....	4
1.4 作用域链.....	4
1.5 形参是局部变量.....	5

1.6 全局变量的应用.....	6
1.7 函数的作用域.....	7
1.8 闭包.....	7
二、数组.....	9
2.1 数组的概述.....	9
2.2 数组的长度.....	9
2.3 数组的遍历.....	10
2.4 数组的收尾操作方法.....	11
2.5 数组的拆分和合并.....	12
2.6 删除，插入，替换.....	13
2.7 倒序和排序.....	14

复习

break:字面含义“找到我需要的东西，不在需要往下进行，结束循环”。

continue: 字面含义“这个结果不是我想要的立刻进行下一个测试”

不管 **break** 还是 **continue** 只能管理自己当前的循环，不能管理外层的循环。如果想控制外层循环需要给一个标签。

函数：

```
1 //函数声明
2 function 函数名(){
3     结构体
4 }
```

```
5 //函数调用
6 函数名();
```

函数表达式:

```
1 var 变量名 = function(){} // 匿名函数。
```

函数声明头的提升。关键字 **function** 这种声明函数可以进行声明头的提升。

函数表达式不能进行函数声明头的提升，只能单纯提升变量名。

一、函数

1.1 递归函数

递归函数是在一个函数通过名字调用自身的情况下构成的。

一个函数可以调用自身，这种现象叫做递归。

```
1 // 递归函数，就是在函数内部调用自身函数
2 function sum(a,b){
3     console.log(sum(a,b) + b);
4 }
5 sum(3,4);
```

应用：可以用递归处理一些数学问题。如斐波那契数列。

斐波那契数列：1,1,2,3,5,8,13,21,34……

```
1 // 书写一个函数求，菲波那切数列的任意一项
2 function feibo(n){
3     for(var i = 1 ; i <= n ; i ++){
4         if(n == 1 || n == 2){
5             return 1;
6         }else{
7             return feibo(n - 1) + feibo(n - 2);
8         }
9     }
10 }
```

```
1
1
2
3
5
8
13
21
6765
832040
```

1.2 变量的作用域

在函数内定义的变量不能从函数之外的任何地方取得，变量仅仅在该函数的内部有定义。

函数就是变量 **a** 的作用域，**a** 只能在他的作用域内被访问到，在作用域外面是不能被访问到的。

块级作用域：{}包裹的就是块级作用域，其他的计算机语言的情况。

JS 语言比较简单，没有块级作用域。js 里，只有函数能够关住作用域。

```
1 // 在函数内部定义的变量不能从函数外部任何地方取得
```

```

2 function fun(){
3     var a = 1;
4     console.log(a);
5 }
6 fun();
7 console.log(a);

```

Uncaught ReferenceError: a is not defined
at 08_变量的作用域.html:15

1.3 局部变量和全局变量

局部变量：在一个作用域（定义域）内定义的变量就是这个作用域内的局部变量。只能在作用域内被访问到。

全局变量：从广义上来看，全局变量也是一种局部变量。全局变量定义在全局，所以也叫全局变量。可以在任何地方都被访问到。

```

1 // b 是在全局内部定义的变量，是全局变量，可以在任何地方访问到
2 var b = 2;
3 function fun(){
4     // 在函数内部定义的变量是局部变量，只能在该作用域内访问到
5     var a = 1;
6     console.log(a);
7 }
8 fun();
9 console.log(b);
10 console.log(a);

```

1
2
Uncaught ReferenceError: a is not defined
at 09_局部变量和全局变量.html:20

变量声明的原理：全局变量，在全局定义之后，会永久存在，任何时候，任何位置访问，都能够找到它。局部变量定义在函数内部的，函数定义的过程，并没有真正的去定义这个局部变量，只有在执行函数的时候，才会立即定义这个局部变量，执行完之后，变量就被立即销毁了，在其他的地方访问变量的时候，找不到这个变量，所以会有一个引用错误，变量未定义。

1.4 作用域链

指的是我们变量查找的一个规律：我们可以在不同的作用域内使用相同的标识符去命名变量。我们在使用一个变量的时候，需要找到匹配的标识符，我们有重复的，用哪一个？如果在当前作用域有这个变量，就直接使用，如果当前作用域没有这个变量定义，会一层一层的从本层往外依次查找，遇到第一个就直接使用。类似于就近原则。

当遇见一个变量时，JS 引擎会从其所在的作用域依次向外层查找，查找会在找到第一个匹配的标识符的时候停止。在多层嵌套的作用域中可以定义同名的标识符，发生“遮蔽效应”。

```

1 // 全局变量
2 var a = 1;
3 function fun1(){
4     var a = 2;
5     function fun2(){

```

```

6      var a = 3;
7      console.log(a);    //因为本层有定义直接输出 3
8      function fun3(){
9          console.log(a); //本身没有 a 定义，会从本层出发依次向外查找，当找到定义时，直接
    执行。3
10     }
11     fun3();
12 }
13 fun2();
14 }
15 fun1();
16 console.log(a);    //1

```

```

3
3
1

```

如果变量声明时，不写 `var` 关键字，计算机会自动在全局作用域内给它进行一个声明，局部变量就强制性的变成了全局变量。这种情况是不合理，会造成一个全局变量的污染。所以，**定义变量必须写 `var` 关键字**。

```

1  var a = 1;
2  // 相当于
3  // var a = 3;
4  function fun1(){
5      a = 2;
6      function fun2(){
7          a = 3;
8          console.log(a);    //因为本层有定义直接输出 3
9          function fun3(){
10             console.log(a); //本身没有 a 定义，会从本层出发依次向外查找，当找到定义时，直接
    执行。3
11         }
12         fun3();
13     }
14     fun2();
15 }
16 fun1();
17 console.log(a);

```

```

3
3
3

```

1.5 形参是局部变量

形参是局部变量，形参的作用域是它定义的**函数的内部**。

```

1  function fun(a){
2      // 形参是局部变量
3      console.log(a);
4  }
5  fun(1);
6  console.log(a);

```



1.6 全局变量的应用

①传递：全局变量可以在不同函数间通信。（相当于信号量）

不同的函数同时控制我们的全局变量。全局变量不会重置也不会清空。

```

1 // 定义一个全局变量
2 var a = 1;
3 // 定义一个自加的函数
4 function jia(){
5     console.log(++a);
6 }
7
8 // 定义一个自减函数
9 function jian(){
10    console.log(--a);
11 }
12
13 jia(); //2
14 jia(); //3
15 jia(); //4
16 jian(); //3
17 jian(); //2
18 jian(); //1
19 jia(); //2
  
```

```

2
3
4
3
2
1
2
  
```

②同一个函数不同调用。累加：全局变量的值，不会被重置、清空。

```

1 //全局变量，不会让变量重置，或者清空
2 var a = 1;
3 function jia(){
4     a += 4;
5     console.log(a);
6 }
7
8
9 // 调用
10 jia(); //5
11 jia(); //9
12 jia(); //13
  
```

5
9
13

1.7 函数的作用域

函数作用域和变量类似，也是只能在函数声明的地方使用，外部任何地方都不能访问。

```
1 function outer(){
2     function inner(){
3         console.log(1);
4     }
5     inner();
6 }
7 outer();
8 inner();    //不能再 outer 的外部调用 inner
```

```
1
▶ Uncaught ReferenceError: inner is not defined
   at 15_函数的作用域.html:14
```

1.8 闭包

体会闭包：

```
1 function outer(){
2     var a = 1;
3     function inner(){
4         console.log(a);
5     }
6     return inner; //没有小括号，表示只输出 inner 函数的定义，不会立即执行
7 }
8 // console.log(outer());
9 var i = outer();
10 // i 内部存储的是 inner 函数的地址。
11 i();
12 // 本层没有 a 的定义，但是 inner 的作用域链存在。
13 // 所以输出 1
```

总结：**inner** 函数把它自己内部的语句，和自己声明时所处的作用域一起封装成了一个密闭环境，我们称为“闭包”。

闭包是天生存在的，并不是我们通过某种方法做出来的。

函数本身就是一个闭包。函数定义的时候，就能记住它的**外部环境**和**内部语句**，每次执行都会参考定义时的密闭环境。

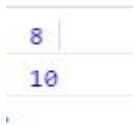


案例 1:

```

1 // 案例 1
2 function outer(x){
3     function inner(y){
4         console.log(x + y);
5     }
6     return inner;
7 }
8 var i = outer(3);
9 // i = function inner(y){
10 // console.log(3+y);
11 // }
12 i(5);
13 i(7);

```



```

8
10

```

说明：初始定义 inner 函数的时候，是在 out（3）执行时，所以 x 记住了一个 3 的值。

闭包天生存在，并不需要什么特殊的结构才存在，只不过我们必须刻意地把函数放到其他的作用域中调用，才能明显的观察到闭包性质。

案例 2:

```

1 function outer(x,y){
2     function inner(x){
3         console.log(x + y);
4     }
5     return inner;
6 }
7
8 var i = outer(2,3);
9 // inner 的外部环境
10 // y = 3;
11 // i = function inner(x){
12 // console.log(x + y);
13 // }
14 i(5);
15 //8

```

案例 3:

```

1 function outer(){
2     var i = 1;
3     function inner(){
4         return i++;
5     }
6     return inner;
7 }
8 var i = outer();
9 console.log(i());
10 console.log(i());
11 console.log(i());

```

结论：函数的闭包，记住了定义时所在的作用域，这个作用域中的变量不是一成不变的。


```

1 function outer(){
2     var i = 1;
3     function inner(){
4         return i++;
5     }
6     return inner;
7 }
8 var i = outer();
9 var inn = outer();
10
11 console.log(i());
12 console.log(inn());
13 console.log(inn());
14 console.log(inn());
15 console.log(i());
16 console.log(i());
17 console.log(i());

```

结论：我们可以认为，**每次调用一个函数，都会产生新的闭包**。新的闭包是指，语句全新，所处环境也是全新的。

二、数组

2.1 数组的概述

数组就是表示一系列有序的数据的**集合**。

数组的表示方法：`[]`。可以这样认为只要看见方括号就是一个数组。

数组中每一个数据之间都用逗号隔开，最后一项没有逗号。

array: 数组

```

1 // 定义一个数组
2 var arr = [1,2,4,5];
3 console.log(arr[2]);

```

使用索引(index)，也称为下标，来精确的**读取、设置**数组中的某一项。数组的下标从**0**开始。

```
1 arr[index]; //读取
```

设置：`arr[2] = 456;`

数组中每一项可以是不同的数据类型。通常我们习惯书写相同的数据类型作为一组数组。

2.2 数组的长度

数组的数据时**引用数据类型**。

```

1 // 数组的数据类型
2 console.log(typeof arr);

```

引用类型一般都有自己的属性和方法；

属性：事物具有的特点。比如：人的高矮胖瘦。

方法：就是事物的行为。比如：玩电脑，睡觉。

我们可以使用**打点**的方法调用属性。

```
1 引用数据.属性
```

数组的长度用 `length` 属性表示。直接打点调用。

```
1 // 得到数组的长度
2 console.log(arr.length);
```

数组的最后一项：下标为数组长度减 1。

```
1 arr[arr.length - 1];
```

如果下标超过 `arr.length - 1`, 值不存在, 输出 `undefined` 未定义。

```
1 var arr = [1,23,45,2,56,1,67,89];
2 // 数组的最大下标是 length -1, 当超过这个下标时输出 undefined
3 console.log(arr[9]);
```

```
undefined
```

我们可以通过下标给某一项赋值。如果我们给下标超过 `arr.length - 1` 的项赋值, **强制赋值, 强制将数组长度拉长了。**

```
1 var arr = [1,23,45,2,56,1,67,89];
2 arr[20] = 20;
3 console.log(arr[20]);
4 // 现在的数组长度是多少
5 console.log(arr.length);
```

```
21
```

虽然长度加长了, 但是中间没有被赋值的项, 还是 `undefined`。

`arr.length` 也可以强制赋值, 如果超过原来的长度, 多出来的部分未赋值就是 `undefined`, 如果少于原来的长度, 会把多出去的下标部分数据直接删除。

```
1 var arr1 = [1,2,45,67,2,5566,67,12];
2 console.log(arr1.length);
3 arr1.length = 5;
4 console.log(arr1.length);
5 console.log(arr1);
```

```
8
```

```
5
```

```
▶ [1, 2, 45, 67, 2]
```

```
>
```

2.3 数组的遍历

我们可以通过某种方法, 得到数组中的每一项。这就是遍历。

我们通过 `for` 循环进行数组的遍历。

```
1 var arr = [1,2,"号","",function() {},12,23,null,false];
2 // 遍历输出数组中每一项
3 for(var i = 0 ; i <= arr.length - 1 ; i++){
4     console.log(arr[i]);
5 }
```

```

1
2
号
function (){}
12
23
null
false

```

案例：arr = [2,4,6,7,8];求数组中每一项的阶乘然后求和。

```

1 // 案例
2 var arr = [2,4,6,7,8];
3 // 求阶乘和
4 // 累加器
5 var sum = 0;
6 for(var i = 0 ; i <= arr.length - 1 ; i ++){
7     // 需要一个求阶乘的函数
8     sum += jiecheng(arr[i]);
9 }
10 console.log(sum);
11
12
13 // 阶乘函数
14 // 累乘器
15 function jiecheng(a){
16     var cheng = 1;
17     for(var i = 1 ; i <= a ; i ++){
18         cheng *= i;
19     }
20     return cheng;
21 }

```

2.4 数组的收尾操作方法

push() 在数组末尾添加一个或多个元素，并返回数组操作后的长度。

pop() 从数组移出最后一个元素，并返回该元素。

shift() 从数组移出第一个元素，并返回该元素。

unshift() 在数组开头添加一个或多个元素，并返回数组的新长度。

语法：数组对象，点语法调用 push 方法，后面加小括号，括号里面的内容就是我们要在末尾添加的元素。

添加的时候，可以添加多项数据，数据之间用逗号隔开即可。

```

1 var arr = [1,2,3,4,5];
2 //push(), 是在最后增加一个或者多个元素，直接把元素写在小括号内，如果有多个数据用逗号隔开。
3 //返回值是新数组的长度
4 console.log(arr.push(6,7,[8,9]));
5 console.log(arr);

```

```

8
▶ [1, 2, 3, 4, 5, 6, 7, Array[2]]
>

```

```

1 //pop:删除数组最后一项，并且返回删除的数据
2 var arr = [1,2,3,4,5];
3 console.log(arr.pop());
4 console.log(arr);

```

```

14 //unshift:在数组开头添加一个或多个元素，并返回数组的新长度。
15 var arr = [1,2,3,4,5];
16 console.log(arr.unshift(0));
17 console.log(arr);

```

```

14 //shift:删除数组第一项，并且返回删除的数据
15 var arr = [1,2,3,4,5];
16 console.log(arr.shift());
17 console.log(arr);

```

案例：

```

1 // 小应用
2 // 把最后一项增加到开头
3 var arr = [1,2,3,4,5];
4 // 增加到开头
5 console.log(arr.unshift(arr.pop()));
6 // [5,1,2,3,4]

```

2.5 数组的拆分和合并

①数组的合并 concat()。

concat()方法返回值是一个新数组，不改变原来的数组。

```

1 var arr1 = [1,2,3,4,5,6];
2 var arr2 = [7,8,9];
3 var arrNew = arr1.concat(arr2);
4 console.log(arrNew);
5 console.log(arr1);

```

```

▶ [1, 2, 3, 4, 5, 6, 7, 8,
  9]

```

```

▶ [1, 2, 3, 4, 5,
  6]

```

concat()参数非常随意，可以是数组，也可以是散值。

```

1 // concat()方法的参数非常随意可以是数组也可以是散值
2 var arr = [1,2,3,4,5,6];
3 var arrNew = arr.concat(7,8,[9,10]);
4 console.log(arrNew);

```

```

▶ [1, 2, 3, 4, 5, 6, 7, 8, 9,
  10]

```

②数组的拆分 slice()。

slice(start,end)。对数组进行拆分返回的是一个新数组。通过数组的 index 进行拆分。start 和 end 用数组下标表示，包括 start 值，不包括 end 值。是一个区间值

```

1 var arr = [1,2,3,4,5,6,7,8];
2 var arrNew = arr.slice(2,5); //从下标 2 开始截取一直到下标为 5。（但是不包含 5）

```

```
3 console.log(arrNew);
4 console.log(arr);
```

```
▶ [3, 4, 5]
▶ [1, 2, 3, 4, 5, 6, 7, 8]
```

下标可以用负值，表示倒数第几项。（倒数是从-1开始）。

```
1 // 下标可以是负值
2 var arr = [1,2,3,4,5,6,7,8];
3 var arrNew = arr.slice(-6,-2); //表示从倒数第6项到倒数第3项
4 console.log(arrNew);
```

```
▶ [3, 4, 5, 6]
```

还可以只写 start 表示截取到最后。

```
1 // 可以不写 end
2 var arr = [1,2,3,4,5,6,7,8];
3 var arrNew = arr.slice(4);
4 console.log(arrNew);
```

```
▶ [5, 6, 7, 8]
```

```
1 // 负值也是可以
2 var arr = [1,2,3,4,5,6,7,8];
3 var arrNew = arr.slice(-6);
4 console.log(arrNew);
```

```
▶ [3, 4, 5, 6, 7, 8]
```

2.6 删除，插入，替换

splice()方法。方法用于插入、删除或替换数组的元素。

语法：arrayObject.splice(index,howmany,element1,...,elementX)

红色：必写的参数。绿色：可选参数。

index：表示删除元素的起始位置的索引值。

howmany：删除的元素个数，如果为0，表示不删除，后面有元素加入，表示插入的意思。如果有个数，表示删除。

element：表示要替换的新的元素。如果有，表示插入或替换，如果没有，只能是删除。

返回值：由被删除的元素组成的一个数组。如果没有删除元素，则返回空数组。原数组发生改变。

```
1 // 删除
2 var arr = [1,2,3,4,5,6,7,8,9];
3 console.log(arr.splice(2,3));
4 // 会改变原数组
5 console.log(arr);
```

```
▶ [3, 4, 5]
▶ [1, 2, 6, 7, 8, 9]
```

```
1 // 插入
2 var arr = [1,2,3,4,5,6,7,8,9];
3 // howmany 是 0, element 必须写
4 console.log(arr.splice(2,0,23,"你好"));
5 console.log(arr);
```

```
▶ []
▶ [1, 2, 23, "你好", 3, 4, 5, 6, 7, 8, 9]
```

```
1 // 替换
2 var arr = [1,2,3,4,5,6,7,8,9];
3 // 返回的是删除的数据
4 console.log(arr.splice(2,3,100,101));
5 console.log(arr);
```

```
▶ [3, 4, 5]
▶ [1, 2, 100, 101, 6, 7, 8, 9]
```

index 可以写负值。

```
1 // index 可以是负值
2 var arr = [1,2,3,4,5,6,7,8,9];
3 console.log(arr.splice(-5,3));
```

```
▶ [5, 6, 7]
```

2.7 倒序和排序

倒序: `reverse()` 颠倒数组元素的顺序: 第一个变成最后一个, 最后一个变成第一个。

```
1 var arr = [1,2,"你好",4,5,null];
2 console.log(arr.reverse());
```

```
▶ [null, 5, 4, "你好", 2, 1]
```

排序: `sort()` 给数组元素排序。

默认情况: 小括号内不传任何参数。

排序依据: 将所有的数组的数据都转成字符串, 然后根据字符编码顺序排序。数字, 大写字母, 小写字母。

```
1 var arr = [12,2,"AB",4,5,null,"x","a"];
2 console.log(arr.sort());
```

```
▶ [12, 2, 4, 5, "AB", "a", null, "x"]
```

`sort` 方法有一个参数叫做比较函数。如果指明了 `compareFunction`, 那么数组会按照调用该函数的返回值排

序。记 a 和 b 是两个将要被比较的元素（自定义升序排列）：

若 a 小于 b ，在排序后的数组中 a 应该出现在 b 之前，则返回一个小于 0 的值。

若 a 等于 b ，则返回 0。

若 a 大于 b ，则返回一个大于 0 的值。

自定义降序排列相反。

```
1 // 自定义降序排序
2 var arr = [12,34,67,23,78,100,3,34];
3
4
5 console.log(arr.sort(function compareFunction(a,b){
6     if(a > b){
7         return -2;
8     }else if(a == b){
9         return 0;
10    }else{
11        return 1;
12    }
13 }));
```

▶ [100, 78, 67, 34, 34, 23, 12, 3]

2.8 join ()

将数组整体转为一个字符串。

如果传递参数，参数会作为连接符，将每一项数据连接起来。

```
1 // join()方法将数组转换为字符串
2 var arr = [1,2,3,4,5,6];
3 // join 传递的参数链接每一项数据
4 console.log(arr.join("+"));
5 console.log(arr.join("文字"));
6 // join 可以不写参数，默认以逗号连接。
7 console.log(arr.join());
```

1+2+3+4+5+6

1文字2文字3文字4文字5文字6

1,2,3,4,5,6

1

1

1

1

1

1

1

1

1

1